```
package server;

import java.io.*;

// User Configuration Information.

public class UserInfo
{
    public Profile          p;                                  // User profile.
    public Screening        sc;                                 // Call screenin¬
g settings.
    public Options          o;                                  // User-selectab¬
le options.
    public MACAddr          ma;                                 // Assigned MACA¬
ddress.
    public ForwardingInfo   fi;                                 // ForwardingInf¬
o

    public static UserInfo  uiNull      =   new UserInfo();     // Initial UserI¬
nfo.


    public UserInfo(Profile p, Screening sc, Options o, MACAddr ma, ForwardingInfo fi)
    {
        this.p      =       p;
        this.sc     =       sc;
        this.o      =       o;
        this.ma     =       ma;
        this.fi     =       fi;
    }

    public UserInfo(Profile p, Screening sc, Options o, MACAddr ma)
    {
        this(p, sc, o, ma, new ForwardingInfo());
    }


    // Default UserInfo.

    public UserInfo(ServerConfig scon)
    {
        this(new Profile(scon), new Screening(), new Options(scon), new MACAddr(), new F¬
orwardingInfo(scon));
    }


    public UserInfo(ServerConfig scon, MACAddr ma)
    {
        this(new Profile(scon), new Screening(), new Options(scon), ma, new ForwardingIn¬
fo(scon));
    }

    public UserInfo(ServerConfig scon, MACAddr ma, String sAltSpoken)
    {
        this(new Profile(scon, sAltSpoken), new Screening(), new Options(scon), ma, new ¬
ForwardingInfo(scon));
    }

    public UserInfo(ServerConfig scon, String sAltSpoken, String sFirst, String sLast)
    {
        this(new Profile(scon, sAltSpoken, sFirst, sLast), new Screening(), new Options(¬
scon), new MACAddr(), new ForwardingInfo(scon));
    }
```

```
    public UserInfo(ServerConfig scon, String sAltSpoken, String sFirst, String sLast, S¬
tring sIdentPhrase)
    {
        this(new Profile(scon, sAltSpoken, sFirst, sLast, sIdentPhrase), new Screening()¬
, new Options(scon), new MACAddr(), new ForwardingInfo(scon));
    }

    public UserInfo(ServerConfig scon, String sEMail, MACAddr ma, String sAltSpoken)
    {
        this(new Profile(scon, sEMail, sAltSpoken), new Screening(), new Options(scon), ¬
ma, new ForwardingInfo(scon));
    }

    public UserInfo(UserInfo ui)
    {
        p       =       ui.p.copy();
        sc      =       ui.sc.copy();
        o       =       ui.o.copy();
        ma      =       ui.ma;
        fi      =       ui.fi.copy();
    }

    public UserInfo()
    {
        this(new Profile(), new Screening(), new Options(), new MACAddr(), new Forwardin¬
gInfo());
    }

    public UserInfo copy()
    {
        return new UserInfo(this);
    }


    public MACAddr  getMACAddr()
    {
        return ma;
    }

    public ForwardingInfo getForwardingInfo()
    {
        return fi;
    }

    // Writes UserInfo to a stream.

    public void write(DataOutputStream s) throws IOException
    {
        p.write(s);
        sc.write(s);
        o.write(s);
        ma.write(s);
        fi.write(s);
    }

    // Reads UserInfo from a stream.

    public static UserInfo read(DataInputStream s) throws IOException
    {
        UserInfo    ui  =   new UserInfo();

        ui.p            =   Profile.read(s);
```

```
        ui.sc           =   Screening.read(s);
        ui.o            =   Options.read(s);
        ui.ma           =   MACAddr.read(s);
        ui.fi           =   ForwardingInfo.read(s);

        return ui;
    }

    public void merge(UserInfo uiOld, UserInfo uiBase)
    {
        (new Merger()).merge(this, uiOld, uiBase);
    }

    // Profile info.

    public static class Profile
    {
        // Permission Masks.

        public final static int pmLocAdmin          =   0x1;                // A¬
dministrator permissions.
        public final static int pmGroupJoin         =   0x2;                // G¬
roup join.
        public final static int pmGroupManage       =   0x4;                // G¬
roup management.
        public final static int pmCallInternal      =   0x8;                // I¬
nternal calls.
        public final static int pmCallExternal      =   0x10;               // E¬
xternal calls.
        public final static int pmVIP               =   0x20;               // V¬
IP call status.
        public final static int pmLocate            =   0x40;               // P¬
ermission to locate users.
        public final static int pmLogin             =   0x80;               // P¬
ermission to log in.
        public final static int pmDefault           =   0xDE;               // D¬
efault permissions.

        public String           sLogin;                                     // L¬
og-in name (could be same as e-mail address).
        public String           sPassword;                                  // P¬
assword.
        public String           sLast;                                      // L¬
ast name.
        public String           sFirst;                                     // F¬
irst name.
        public StringSet        ssAltSpoken;                                // A¬
lternative spoken names.
        public String           sIdentPhrase;                               // I¬
dentifying phrase.
        public String           sEMail;                                     // E¬
-mail address.
        public String           sWorkPhone;                                 // W¬
ork phone no (or extension).
        public String           sCellPhone;                                 // C¬
ell phone no.
        public String           sHomePhone;                                 // H¬
ome phone no.
        public int              iPermissions;                               // P¬
ermission bit vector.
        public String           sPIN;                                       // P¬
in code for outside calls.
        public String           sOwner;                                     // O¬
```

wing user for a personal contact.

```
        public Profile()
        {
            sLogin              =       "";
            sPassword           =       "";
            sLast               =       "";
            sFirst              =       "";
            ssAltSpoken         =       new StringSet();
            sIdentPhrase        =       "";
            sEMail              =       "";
            sWorkPhone          =       "";
            sCellPhone          =       "";
            sHomePhone          =       "";
            iPermissions        =       pmDefault;
            sPIN                =       "";
            sOwner              =       "";
        }

        public Profile(int iDefaultPermissions)
        {
            this();

            this.iPermissions   =   iDefaultPermissions;
        }


        public Profile(ServerConfig scon)
        {
            this();

            iPermissions        =   scon.getDefaultPermissions();
        }

        // Copy constructor.

        public Profile(Profile p)
        {
            sLogin          =   p.sLogin;
            sPassword       =   p.sPassword;
            sLast           =   p.sLast;
            sFirst          =   p.sFirst;
            ssAltSpoken     =   p.ssAltSpoken.copy();
            sIdentPhrase    =   p.sIdentPhrase;
            sEMail          =   p.sEMail;
            sWorkPhone      =   p.sWorkPhone;
            sCellPhone      =   p.sCellPhone;
            sHomePhone      =   p.sHomePhone;
            iPermissions    =   p.iPermissions;
            sPIN            =   p.sPIN;
            sOwner          =   p.sOwner;
        }

        public Profile(ServerConfig scon, String sAltSpoken)
        {
            this(scon);

            if (!sAltSpoken.equals(""))
                ssAltSpoken.add(sAltSpoken);
        }

        // Temporary constructor for test purposes.
```

```
        public Profile(ServerConfig scon, String sAltSpoken, String sFirst, String sLast¬
)
        {
            this(scon, sAltSpoken, sFirst, sLast, "");
        }

        public Profile(ServerConfig scon, String sAltSpoken, String sFirst, String sLast¬
, String sIdentPhrase)
        {
            this(scon);

            this.sFirst         =       sFirst;
            this.sLast          =       sLast;
            this.sIdentPhrase   =       sIdentPhrase;

            sLogin              =       makeLogin(sFirst, sLast);
            sEMail              =       sLogin + "@vocera.com";

            if (!sAltSpoken.equals(""))
                ssAltSpoken.add(sAltSpoken);
        }

        // Temporary, for dinking up a login name.

        private String makeLogin(String sFirst, String sLast)
        {
            String  sFirstLetter        =   sFirst.equals("") ? "" : ("" + sFirst.charAt¬
(0));

            return (sFirstLetter + sLast).toLowerCase();
        }

        public Profile(ServerConfig scon, String sEmail, String sAltSpoken)
        {
            this(scon);
            this.sEMail         =       sEmail;

            if (!sAltSpoken.equals(""))
                ssAltSpoken.add(sAltSpoken);
        }

        public Profile copy()
        {
            return new Profile(this);
        }

        // Returns set of spoken pronunciations.

        public StringSet getSpokenNames()
        {
            StringSet   ss              =       new StringSet();
            String      sSpokenName     =       getSpokenName();

            if (!sSpokenName.equals(""))
                ss.add(getSpokenName());                                        // A¬
dd primary spoken name.

            ss.add(ssAltSpoken);

            if (sIdentPhrase.length() > 0)
                ss.add(sIdentPhrase);

            return ss;
```

```
    }

    // Returns primary spoken name.

    public String getSpokenName()
    {
        if (sFirst.equals(""))
            return sLast.toLowerCase();

        if (sLast.equals(""))
            return sFirst.toLowerCase();

        return sFirst.toLowerCase() + " " + sLast.toLowerCase();
    }

    // Returns set of allowed spellings.

    public StringSet getSpelledNames()
    {
        StringSet   ss      =   new StringSet();

        if (sLast.length() > 0)
            ss.add(sLast);

        return ss;
    }

    // Returns true if has permissions given by iMask.

    public boolean hasPermissions(int iMask)
    {
        return (iMask & iPermissions) == iMask;
    }

    // Enables or disables permissions associated with given mask;

    public void setPermissions(int iMask, boolean bEnable)
    {
        if (bEnable)
            iPermissions    |=  iMask;
        else
            iPermissions &= ~iMask;
    }

    // Returns true for a personal contact.

    public boolean isPersonal()
    {
        return sOwner.length() > 0;
    }

    // Writes profile out to a streasm.

    public void write(DataOutputStream s) throws IOException
    {
        s.writeUTF(sLogin);
        s.writeUTF(sPassword);
        s.writeUTF(sLast);
        s.writeUTF(sFirst);
        ssAltSpoken.write(s);
        s.writeUTF(sIdentPhrase);
        s.writeUTF(sEMail);
        s.writeUTF(sWorkPhone);
```

```
            s.writeUTF(sCellPhone);
            s.writeUTF(sHomePhone);
            s.writeInt(iPermissions);
            s.writeUTF(sPIN);
            s.writeUTF(sOwner);
        }

        public static Profile read(DataInputStream s) throws IOException
        {
            Profile     p       =   new Profile();

            p.sLogin                =   s.readUTF();
            p.sPassword             =   s.readUTF();
            p.sLast                 =   s.readUTF();
            p.sFirst                =   s.readUTF();
            p.ssAltSpoken           =   StringSet.read(s);
            p.sIdentPhrase          =   s.readUTF();
            p.sEMail                =   s.readUTF();
            p.sWorkPhone            =   s.readUTF();
            p.sCellPhone            =   s.readUTF();
            p.sHomePhone            =   s.readUTF();
            p.iPermissions          =   s.readInt();
            p.sPIN                  =   s.readUTF();
            p.sOwner                =   s.readUTF();

            return p;
        }
    }

    // Screening information.

    public static class Screening
    {
        // Call Screening Settings.

        public final static int     cscBlockAll                 =   0;
        public final static int     cscAllowAll                 =   1;

        public int                  iCallScreening;
// Currently-selected setting.
        public BuddySet             bsBuddies;
// Set of buddies.
        public StringSet            ssExceptions;


        public Screening(int iCallScreening, BuddySet bsBuddies, StringSet ssExceptions)
        {
            this.iCallScreening     =   iCallScreening;
            this.bsBuddies          =   bsBuddies;
            this.ssExceptions       =   ssExceptions;
        }

        public Screening(Screening sc)
        {
            iCallScreening          =   sc.iCallScreening;
            bsBuddies               =   sc.bsBuddies.copy();
            ssExceptions            =   sc.ssExceptions.copy();
        }


        public Screening()
        {
            this(cscAllowAll, new BuddySet(), new StringSet());
```

```
        }

        public Screening copy()
        {
            return new Screening(this);
        }

        // Returns true if sUserName is the user name of a buddy, either directly or as a member of a group.

        public boolean isBuddy(Server s, String sUserName)
        {
            return Entity.getEntities(s, bsBuddies.getNames()).expand().getNames().contains(sUserName);
        }

        // Returns true if sName is the entity name of a buddy, not indirectly as a member of a group.

        public boolean isImmediateBuddy(String sName)
        {
            return bsBuddies.findName(sName) != -1;
        }

        // Returns true if sName is the entity name of a screening exception, not indirectly as a member of a group.

        public boolean isImmediateException(String sName)
        {
            return ssExceptions.find(sName) != -1;
        }

        // Returns true if sUserName is the user name of a buddy with VIP status.

        public boolean isVIPBuddy(Server s, String sUserName)
        {
            return bsBuddies.isVIPBuddy(s, sUserName);
        }

        // Returns true if currently blocking calls from party with given name.

        public boolean isBlocked(Server s, String sName)
        {
            return (iCallScreening == cscBlockAll) ^ isException(s, sName);
        }

        // Returns true if sName is an exception, either directly or as a member of an exception group.

        private boolean isException(Server s, String sName)
        {
            return Entity.getEntities(s, ssExceptions).expand().getNames().contains(sName);
        }

        // Returns true if this is equal to given Screening.

        public boolean isEqual(Screening sc)
        {
            if (iCallScreening != sc.iCallScreening)
                return false;

            if (!bsBuddies.getNames().isEqual(sc.bsBuddies.getNames()))
```

```
            return false;

        if (!ssExceptions.isEqual(sc.ssExceptions))
            return false;

        return true;
    }

    // Updates Screening information according to given parameters.

    public void update(boolean bBlock, boolean bOnly, boolean bExcept, boolean bBudd¬
ies, StringSet ssUsers)
    {
        if (bBlock)
            block(bOnly, bExcept, bBuddies, ssUsers);
        else
            allow(bOnly, bExcept, bBuddies, ssUsers);
    }

    private void block(boolean bOnly, boolean bExcept, boolean bBuddies, StringSet s¬
sUsers)
    {
        if (bOnly)                                          // Block only case.
        {
            setScreening(cscAllowAll);
            if (bBuddies)
                addBlock(bsBuddies.getNames());
            addBlock(ssUsers);
        }
        else
        if (bExcept)                                        // Block all except case.
        {
            setScreening(cscBlockAll);
            addAllow(ssUsers);
        }
        else
        if (!bBuddies && ssUsers.size() == 0)
            setScreening(cscBlockAll);
        else
        {
            if (bBuddies)
                addBlock(bsBuddies.getNames());
            addBlock(ssUsers);
        }
    }

    private void allow(boolean bOnly, boolean bExcept, boolean bBuddies, StringSet s¬
sUsers)
    {
        if (bOnly)
            block(false, true, bBuddies, ssUsers);
        else
        if (bExcept)
            block(true, false, bBuddies, ssUsers);
        else
        if (!bBuddies && ssUsers.size() == 0)
            setScreening(cscAllowAll);
        else
        {
            if (bBuddies)
                addAllow(bsBuddies.getNames());
            addAllow(ssUsers);
        }
```

```
    }

    private void setScreening(int iCallScreening)
    {
        this.iCallScreening     =   iCallScreening;
        ssExceptions.empty();
    }

    // Incrementally blocks ssUsers.

    private void addBlock(StringSet ssUsers)
    {
        switch (iCallScreening)
        {
            case cscBlockAll:
                ssExceptions.remove(ssUsers);
                break;

            case cscAllowAll:
                ssExceptions.add(ssUsers);
                break;

            default:
                Debug.fail();
        }
    }

    // Incrementally allows ssUsers.

    private void addAllow(StringSet ssUsers)
    {
        switch (iCallScreening)
        {
            case cscBlockAll:
                ssExceptions.add(ssUsers);
                break;

            case cscAllowAll:
                ssExceptions.remove(ssUsers);
                break;

            default:
                Debug.fail();
        }
    }

    public void write(DataOutputStream s) throws IOException
    {
        s.writeInt(iCallScreening);
        bsBuddies.write(s);
        ssExceptions.write(s);
    }

    public static Screening read(DataInputStream s) throws IOException
    {
        Screening   sc      =   new Screening();

        sc.iCallScreening   =   s.readInt();
        sc.bsBuddies        =   BuddySet.read(s);
        sc.ssExceptions     =   StringSet.read(s);

        return sc;
    }
```

```
    }

    // User-selectable options.

    public static class Options
    {
        // Boolean options bitmasks.

        // NOTE:  Do not change these without changing boDefault in b\se.h.

        public final static int     boNone                      =   0x0;
// None.
        public final static int     boVerbalCallAnnouncement    =   0x1;
// True if caller's name is to be spoken.
        public final static int     boVerbalGenieGreeting       =   0x2;
// True if genie is announced verbally.
        public final static int     boTonalGenieGreeting        =   0x4;
// True if genie is announced by a tone.
        public final static int     boAutoAnswer                =   0x8;
// Auto-answer option.
        public final static int     boAutoWhoCalled             =   0x10;
// Auto-who called option.
        public final static int     boOutOfRangeAlert           =   0x20;
// Audible alert on out-of-range.
        public final static int     boLowBatteryAlert           =   0x40;
// Audible alert on battery low.
        public final static int     boAutoLogOut                =   0x80;
// Clears badge assignment upon on charger detect.
        public final static int     boTour                      =   0x100;
// True if currently in access point tour mode.
        public final static int     boLostBadgeFinder           =   0x200;
// True if currently in "lost badge finder" mode.
        public final static int     boVMessageAlert             =   0x400;
// Audible alert on text message waiting.
        public final static int     boTMessageAlert             =   0x800;
// Audible alert on voice message waiting.
        public final static int     boDisableAlertsInDND        =   0x1000;
// True if should disable alerts when in DND mode.

        // NOTE:  Do not change this without changing boDefault in b\se.h.

        public final static int     boDefault                   =    0xC5F;
// Everything but AutoLogout/Tour/Badgefinder/OutOfRange/DisAlertsDND
        public final static String  sDefaultRingTone            =   "Ring-Tone-1";
        public final static String  sDefaultPersona             =   "Jean";

        public int                  iOptions;
// Boolean options bit vector.
        public String               sRingTone;
// Ring tone choice.
        public String               sGeniePersona;
// Genie personality options.

        public Options(int iOptions, String sRingTone, String sGeniePersona)
        {
            this.iOptions           =   iOptions;
            this.sRingTone          =   sRingTone;
            this.sGeniePersona      =   sGeniePersona;
        }

        public Options()
        {
            this(0, "", "");
```

```
    }

    public Options(Options o)
    {
        iOptions            =   o.iOptions;
        sRingTone           =   o.sRingTone;
        sGeniePersona       =   o.sGeniePersona;
    }


    // Default options constructor.

    public Options(ServerConfig scon)
    {
        iOptions                    =   scon.getDefaultOptions();
        sRingTone                   =   scon.getDefaultRingTone();
        sGeniePersona               =   scon.getDefaultPersona();
    }


    public Options copy()
    {
        return new Options(this);
    }

    // Returns true if all options in iMask are set.

    public boolean hasOptions(int iMask)
    {
        return (iOptions & iMask) == iMask;
    }


    // Sets/Resets given options.

    public void setOptions(int iMask, boolean bSet)
    {
        if (bSet)
            iOptions    |=  iMask;
        else
            iOptions    &=  ~iMask;
    }

    // Writes options out to a stream.

    public void write(DataOutputStream s) throws IOException
    {
        s.writeInt(iOptions);
        s.writeUTF(sRingTone);
        s.writeUTF(sGeniePersona);
    }

    public static Options read(DataInputStream s) throws IOException
    {
        Options     o               =       new Options();

        o.iOptions                  =       s.readInt();
        o.sRingTone                 =       s.readUTF();
        o.sGeniePersona             =       s.readUTF();

        return o;
    }
}
```

```
    public static class ForwardingInfo
    {
        public final static int     foNone                  =   0;
// No forwarding.
        public final static int     foDeskPhone             =   1;
// Forward to desk phone.
        public final static int     foCellPhone             =   2;
// Forward to cell phone.
        public final static int     foHomePhone             =   3;
// Forward to home phone.
        public final static int     foVoiceMail             =   4;
// Forward to voice mail.
        public final static int     foOtherPhone            =   5;
// Forward to other phone number.
        public final static int     foBadge                 =   6;
// Forward to another badge.

        // When to forward option.

        public final static int     fwNever                 =   0x0;
// Applies only if foNone selected.
        public final static int     fwOffline               =   0x1;
// Forward if user is not logged in or not on network.
        public final static int     fwBusy                  =   0x2;
// Forward if user is currently busy with another call.
        public final static int     fwRefused               =   0x4;
// Forward if call is blocked or refused.
        public final static int     fwAlways                =   0x8;
// Always forward, even if user is available.

        public final static int     fwDefault[]     =
        {
            fwNever,
            fwOffline,
            fwOffline,
            fwOffline,
            fwOffline | fwBusy | fwRefused,
            fwOffline,
            fwOffline | fwBusy | fwRefused
        };

        public int                  iForwarding;
// One of forwarding types above.
        public String               sForwardingNo;
// Forwarding phone no for foPhone.
        public String               sForwardingName;
// Forwarding entity for foBadge.
        public int[]                iForwardWhen;
// When to forward for each option.

        public ForwardingInfo()
        {
            this(foNone, "", "", fwDefault);
        }

        public ForwardingInfo(int iForwarding, String sForwardingNo, String sForwardingName, int[] iForwardWhen)
        {
            this.iForwarding            =   iForwarding;
            this.sForwardingNo          =   sForwardingNo;
            this.sForwardingName        =   sForwardingName;
            this.iForwardWhen           =   iForwardWhen;
```

```
    }

    // Copy constructor.

    public ForwardingInfo(ForwardingInfo fi)
    {
        iForwarding         =   fi.iForwarding;
        sForwardingNo       =   fi.sForwardingNo;
        sForwardingName     =   fi.sForwardingName;

        iForwardWhen        =   new int[fwDefault.length];

        for (int i = 0; i < iForwardWhen.length; ++i)
            iForwardWhen[i]     =   fi.iForwardWhen[i];
    }

    // Default forwarding constructor.

    public ForwardingInfo(ServerConfig scon)
    {
        this();
    }

    public ForwardingInfo copy()
    {
        return new ForwardingInfo(this);
    }

    public boolean equals(Object o)
    {
        return (o instanceof ForwardingInfo) && isEqual((ForwardingInfo) o);
    }

    public boolean isEqual(ForwardingInfo fi)
    {
        if (iForwarding != fi.iForwarding)
            return false;

        if (!sForwardingNo.equals(fi.sForwardingNo))
            return false;

        if (!sForwardingName.equals(fi.sForwardingName))
            return false;

        for (int i = 0; i < iForwardWhen.length; ++i)
        {
            if (iForwardWhen[i] != fi.iForwardWhen[i])
                return false;
        }

        return true;
    }

    // Writes this to a stream.

    public void write(DataOutputStream s) throws IOException
    {
        s.writeInt(iForwarding);
        s.writeUTF(sForwardingNo);
        s.writeUTF(sForwardingName);
        writeIntArray(s, iForwardWhen);
    }
```

```
        private static void writeIntArray(DataOutputStream s, int[] ia) throws IOExcepti¬
on
        {
            s.writeInt(ia.length);

            for (int i = 0; i < ia.length; ++i)
                s.writeInt(ia[i]);
        }

        // Reads in from a stream.

        public static ForwardingInfo read(DataInputStream s) throws IOException
        {
            int     iForwarding     =   s.readInt();
            String  sForwardingNo   =   s.readUTF();
            String  sForwardingName =   s.readUTF();
            int[]   iForwardWhen    =   readIntArray(s);

            return new ForwardingInfo(iForwarding, sForwardingNo, sForwardingName, iForw¬
ardWhen);
        }

        private static int[] readIntArray(DataInputStream s) throws IOException
        {
            int[]       ia      =   new int[s.readInt()];

            for (int i = 0; i < ia.length; ++i)
                ia[i]   =   s.readInt();

            return ia;
        }
    }

    // Three-way merger.

    private static class Merger extends ThreeWayMerger
    {
        static String sBitVectors[]     =
        {
            "iOptions", "iPermissions"
        };

        public Merger()
        {
            super(sBitVectors);
        }

        protected boolean isAtomic(Class c)
        {
            return super.isAtomic(c) || c == ForwardingInfo.class || c == BuddySet.class¬
;
        }

    }
}
```